SI 335, Unit 7: Advanced Sort and Search

Daniel S. Roche (roche@usna.edu)

Spring 2015

Sorting's back! After starting out with the sorting-related problem of computing medians and percentiles, in this unit we will see two of the most practically-useful sorting algorithms: quickSort and radixSort. Along the way, we'll get more practice with divide-and-conquer algorithms and computational models, and we'll touch on how random numbers can play a role in making algorithms faster too.

1 The Selection Problem

1.1 Medians and Percentiles

Computing statistical quantities on a given set of data is a really important part of what computers get used for. Of course we all know how to calculate the *mean*, or *average* of a list of numbers: sum them all up, and divide by the length of the list. This takes linear-time in the length of the input list.

But averages don't tell the whole story. For example, the average household income in the United States for 2010 was about $68,000 but the median was closer to $50,000. Somehow the second number is a better measure of what a "typical" household might bring in, whereas the first can be unduly affected by the very rich and the very poor.

Medians are in fact just a special case of a *percentile*, another important statistical quantity. Your SAT scores were reported both as a raw number and as a percentile, indicating how you did compared to everyone else who took the test. So if 100,000 people took the SATs, and you scored in the 90th percentile, then your score was higher than 90,000 other students' scores. Another way of putting this is that your score was higher than the 90,000th score.

Since this is an algorithms class, hopefully you see where we are going: we want to think about the best way to actually compute these values. Given a list of 100,000 numbers, how can we find the 50,000th largest one (the median), or the 90,000th largest one (the 90th percentile) or even the 100,000th largest one (the max)? These are all special cases of the following general problem for our consideration:

> **Selection Problem**
>
> Input: A list of values A of length n, and an integer k satisfying $0 \leq k < n$
>
> Output: The kth largest value in A, counting from zero.

Observe that, as good computer scientists, we are counting from zero! So the 0th largest value is the smallest one, and the $(n-1)$th largest is the maximum. This conflicts with the normal English usage, but it'll be easier for us to talk about the problem this way.

As we delve into this problem, don't forget the big picture question: can we compute the median of a list in the same time it takes to compute the average, or is finding the median somehow inherently more difficult? Are we going to come up with a brilliant algorithm or a brilliant lower bound for the problem? Take a second to see what you could come up with before we continue.

1.2 Sorting-based solutions

At least one solution to this problem should be glaringly obvious: just sort the list A and then return the value at index k from the sorted list. Using an asymptotically optimal algorithm such as HeapSort or MergeSort, this costs $O(n \log n)$ time.

```
def selectBySort(A, k):
    mergeSort(A)
    return A[k]
```

This solution should feel unsatisfying, because it clearly performs a lot of extra work. If we want the median, we are indeed asking what element shows up in the middle of the sorted list, but we really don't care about the order of the first half or the second half of that list. So somehow it seems that sorting the whole list is doing a lot of unnecessary work.

At least when k is very small, we should be able to do better. We all know how to find the smallest element in a list, and this takes only $O(n)$ time. To find the 5th smallest, say, one option would be to take the smallest element, remove it, then continue (four more times) until the smallest element in the remaining list is the 5th smallest in the original list.

This is actually not really a new idea; it's just a short-circuited selection sort! To find the kth smallest element in a list, simply perform the first k steps of the SelectionSort algorithm (to sort the first k elements) and then return the smallest element in the remaining list. This approach has worst-case cost $O(kn)$, which is much better when k is very small (like a constant), but worse if for example we want to find the median.

1.3 Heap-based solutions

If I asked you to find the second-shortest person in the room, you would probably not pick out the shortest first, then start all over and find the shortest person out of everyone remaining. A much better approach would be to keep track of the *two shortest* people as you go around the room, and then take the taller of those two at the end.

How can we generalize this to any k? Well, we want to keep track of the k smallest values in the list as we look through it in a single loop. At each step, we want to insert the next element into our k-smallest structure (so now it has size $k+1$), then then remove the largest of those to bring it back to size k.

What data structure can support insertions and remove-max operations? Why a priority queue of course! In particular, a max-heap will work very nicely to store the k smallest elements. Since the heap has size at most k, every operation on it will cost $O(\log k)$, for a total cost (looping over all the list elements) of $\Theta(n \log k)$.

Now this is truly superior to the two sorting-based algorithms, whether k is small or large. At least it can never be worse than either of those two algorithms. But consider the original case we were interested in: finding the median. Then $k = \lfloor n/2 \rfloor$, so the asymptotic cost is once again $\Theta(n \log n)$ - the same as sorting! Can we do better?

Let's try short-circuiting the HeapSort algorithm, just like we short-circuited SelectionSort above. Remember that the HeapSort algorithm just does a single heapify operation on the whole array, then calls removeMax on the heap n times to get all the elements in sorted order.

So to solve the selection problem, we could heapify our array into a min-heap, then call removeMin k times, returning the last removed value which must be the kth smallest element. Since heapify can be performed in $\Theta(n)$ time, the total cost of this approach is $\Theta(n + k \log n)$. Here it is in pseudocode:

```
def selectByHeap(A, k):
    H = copy(A)
    heapify(H)
    for i in range(0, k):
        heappop(H) # this is the remove-min operation.
    return H[0] # H[0] contains the minumum element in H
```

Is this better than the $\Theta(n \log k)$ algorithm above? Absolutely! For example, say $k \in O(\sqrt{n})$. Then the algorithm above costs $\Theta(n \log \sqrt{n})$, which is $\Theta(n \log n)$, whereas this version is just $\Theta(n + \sqrt{n} \log n)$, which is $\Theta(n)$!

In fact, there are some other tricks we could use with the heap to make this algorithm run in time $\Theta(n + k \log k)$. This will be a very effective algorithm when the element we are searching for is close to the smallest (or largest) in the array. But unfortunately, for our original problem of finding the median, it's still $\Theta(n \log n)$ — no better than sorting. We seem to have hit a wall...

2 QuickSelect

All the solutions above to the selection problem represent an important idea that we should always try when we see a new problem: *reduction to known problem or data structure*. As you encounter all kinds of computational problems in this class and elsewhere, the first thing to do is always to try and tackle the new problem using ideas, algorithms, and data structures that you already know about. For most problems, a completely new idea is not really required to get a fantastic solution.

But for the selection problem, we do need a new idea to improve on the heap-based algorithms. Fortunately, the new idea is not really that new: it's divide-and-conquer all over again.

Ultimately, we would love to have a solution like BinarySearch: split the original array into two parts, only one of which contains the desired answer, and then recurse on one of those two parts. But here of course the input array is *not* pre-sorted, and if we just split the array in half we wouldn't know which half the answer was in, let alone what the value of k should be for the recursive call!

So we are going to have to do more work in the splitting phase. This is going to require a new and very important helper function:

2.1 Partition

The problem we are trying to solve (selection) could be phrased as: given a position (the index k), find the element in that position in the sorted array. Now try turning this around: given an element in the array, find the position of that element in the sorted array.

Now solving the turned-around problem is actually pretty easy: just loop through the array, comparing every element to the given one, and count how many are smaller than it. This count is exactly the index of that element in the sorted array.

The partition function follows exactly this approach and also does a bit more. Besides just finding the position of the given element in the sorted array, it actually puts it there, with everything less than that element coming before it, and everything greater than it coming after. This crucial searched-for element is called the *pivot*, and what the partition function is really doing is *partitioning* the input array into three parts: everything smaller than the pivot, followed by the pivot itself, followed by everything larger than the pivot.

Here's the algorithm:

```
def partition(A):
    '''Partitions␣A␣according␣to␣A[0].␣A[0]␣is␣used␣as␣the␣pivot,
␣␣␣␣␣␣␣␣and␣the␣final␣index␣where␣A[0]␣ends␣up␣(p)␣is␣returned.'''
    n = len(A)
    i, j = 1, n-1
    while i <= j:
        if A[i] <= A[0]:
            i = i + 1
        elif A[j] > A[0]:
            j = j - 1
        else:
            swap(A, i, j)
    swap(A, 0, j)
    return j
```

The idea of this algorithm is to go through the array from both ends simultaneously, swapping pairs of wrongly-positioned elements as we come to them.

Now we'd like to do some analysis on this algorithm. First, let's prove that it's correct. What tool will we use to do this? A loop invariant of course! Here's one that will work:

Every element before A[i] is less than or equal to A[0], and every element after A[j] is greater than A[0].

Remember the three parts to a loop invariant proof? Here they are:

- **Initialization**: At the beginning, the only element before A[i] is A[0], and there aren't any elements after A[n−1], so the invariant is true initially.
- **Maintenance**: Whenever i increases or j decreases, the element that it moves past satisfies the invariant condition, so the invariant is maintained. And if there is swap, it is between elements that are not before A[i] or after A[j], so again the invariant is maintained.
- **Termination**: At the end, the pivot is moved into A[j], and the invariant tells us that everything after A[j] is greater than the pivot, as required. Also, since $i > j$ when the loop finishes, the invariant tells us that everything in A[0..j] is less than or equal to the pivot, as required.

This tells us that the algorithm always returns the correct things, but how do we know it returns at all? Let's do the run-time analysis, which will include a proof of termination.

Just like the analysis of binarySearch, the key is to look at the value of the difference $j - i$. Initially this is equal to $n - 2$, it never increases, and when it decreases below 0 the loop will terminate. The problem here is that pesky **else** case where i and j don't change. Fortunately, we can see that if that case is true, then after the swap, the condition of the **if** will no longer be false. In other words, *we never do two swaps in a row.* And every time we don't swap, the value of *j*-*i* decreases by exactly one.

So in the worst case, the number of iterations through the loop is $2(n - 1)$, which is $\Theta(n)$. And since everything else is just a single primitive operation, this is the total worst-case cost of the algorithm.

2.2 Completing the algorithm

So the idea of the (hopefully) improved selection algorithm we're developing is to first partition the input array into two parts, then make a recursive call on whichever part has the kth element that we are looking for.

Now in order for this to be a good divide-and-conquer algorithm, we need the size of each partitioned half to be close to the same. What's going to determine this is the choice of the pivot element every time. So the very best choice of a pivot would be a median element of the array. But that's the whole problem we're trying to solve!

Well for starters at least we'll just do the dumbest possible thing: choose the first element of the array as the pivot. Improving this choice of the pivot will be a topic of continued discussion as we go along.

Here is the initial (dumb) pivot-selection algorithm:

```
def choosePivot1(A):
    return 0
```

And now we can present the QuickSelect algorithm in all its glory:

```
def quickSelect1(A, k):
    n = len(A)
    swap(A, 0, choosePivot1(A))
    p = partition(A)
    if p == k:
        return A[p]
    elif p < k:
        return quickSelect1(A[p+1 : n], k-p-1)
    elif p > k:
        return quickSelect1(A[0 : p], k)
```

3 Analysis of QuickSelect

So the question is, how fast is our fancy new algorithm? This turns out to be a rather challenging question, and will eventually require a new kind of analysis.

3.1 Best-case

We haven't really done much *best-case* analysis yet. But the different cases of the quickSelect algorithm — based mostly on the choice of pivot — end up being quite interesting, so let's examine the full realm of possibilities here.

In the very best case, the pivot we choose initially could just be the exact element we're looking for! Then we will just have to do one partitioning, and no recursive calls, giving a total best-case cost of $\Theta(n)$.

3.2 Worst-case

If the pivot is the maximum or minimum element in the list, then the two sub-lists that we partition into will have size 0 and $n-1$, respectively, and there will be a recursive call on a list with only one element removed.

So the worst-case cost is given by

$$T(n) = \begin{cases} 1, & n = 1 \\ n + T(n-1), & n \geq 2 \end{cases}$$

From Master Method B, we know that this is $\Theta(n^2)$. Not very good!

3.3 Average-case

We have seen that the best-case of quickSelect is linear-time, but the worst-case is quadratic. We've actually seen an algorithm like this before: insertion sort. But if quickSelect really like a quadratic-time sorting algorithm, then we haven't gained anything at all! We would be much better off using a heap-based selection algorithm.

Actually, this is a case where the worst-case analysis is rather misleading. The reason is that *the worst case is very rare*. Sure, we might get a bad choice for the pivot once or twice, but having a bad choice every time seems unlikely.

So let's analyze how much quickSelect costs *on average*. What this means is that we'll assume every possible input is equally likely, and calculate the average of all the possible costs. For selection, as with sorting, the cost really just depends on the relative ordering of the elements in the input array. To do the average-case analysis, then, we will assume that each of the $n!$ permutations of the array A are equally likely. The average cost, or *average-case analysis*, will be the total cost for all of the $n!$ permutations, added together, then divided by $n!$.

Measuring the cost of every possible permutation seems like a daunting task. Instead of doing that, let's simplify a bit by identifying the real difference-maker in the cost of the algorithm. For quickSelect, it's the *position of the pivot element*, which in the algorithm is p.

Define $T(n,k)$ to be the average cost of quickSelect1(A,k). We can split this into three cases, based on the position of the first-chosen pivot element:

$$T(n,k) = \begin{cases} n + T(n-p-1, k-p-1), & p < k \\ n, & p = k \\ T(p,k), & p > k \end{cases}$$

Now notice that, among all the possible permutations, since we are always choosing the very first element to be the pivot, each of the n possible positions for the pivot element occurs $(n-1)!$ times. Since there are $n!$ total permutations, this means that the chance of the pivot being at position p, for any p, is exactly $\frac{1}{n}$.

Therefore we can write the average cost as the sum of the probability of each case, times the cost of each case, which comes out to

$$T(n,k) = n + \frac{1}{n}\left(\sum_{p=0}^{k-1} T(n-p-1, k-p-1) + \sum_{p=k+1}^{n-1} T(p,k)\right)$$

If you look in your book, you will see a very difficult, detailed analysis of this function.

3.4 Average-case, a little simpler

Instead, we'll look at a simplified version which gives exactly the same *asymptotic* result. Notice that we have already simplified the $n!$ original possibilities to just n equally-likely positions for the pivot element. Now we'll simplify these n possibilities down to only two:

- **Good pivot**: We'll say the pivot is "good" if it falls in the middle half of the sorted array. That is, if $\frac{n}{4} \le p < \frac{3n}{4}$.
- **Bad pivot**: Otherwise, if the pivot is in the bottom quarter of the order, or the top quarter, it is "bad". This means that either $p < \frac{n}{4}$ or $p \ge \frac{3n}{4}$.

Now since each of these represents half of the possible pivot positions, and each of those is equally likely, then the two possibilities of "good pivot" or "bad pivot" are also equally likely. Now let's come up with an upper bound on the cost in each of these two cases:

$$T(n) \le \begin{cases} n + T(\frac{3n}{4}), & \text{good pivot} \\ n + T(n), & \text{bad pivot} \end{cases}$$

See how that worked? The important thing is that, if we have a good pivot, then *no matter what k is*, the recursive call will always be on an array that has size at most three-fourths of the original array size. If we have a bad pivot, then of course all bets are off. You should also notice that we have effectively taken k out of the equation, so it doesn't matter if we're looking for the max or the median or anything in-between. (We could actually have $T(n-1)$ in the recurrence for the "bad pivot" case, but using $T(n)$ instead still gives an upper bound and will make the analysis easier.)

Since we have just two possibilities, and each occurs with one-half probability, the total average-case cost is now:

$$T(n) \le \frac{1}{2}(n + T(\frac{3n}{4})) + \frac{1}{2}(n + T(n))$$

To simplify this, we multiply both sides by 2, then combine like terms and subtract $T(n)$ from both sides to obtain

$$T(n) \le 2n + T(\frac{3n}{4})$$

Hooray! This finally looks like something that we can handle. In fact, we can apply the Master Method A to it, which gives us $e = (\lg a)/(\lg b) = (\lg 1)/(\lg(4/3)) = 0$. Therefore $e < c = 1$, so the total cost is $T(n) \in O(n)$.

Notice that this is just a big-O upper bound. But we already know that the best case of this algorithm is $\Theta(n)$, so therefore the average-case is actually $\Theta(n)$.

This is a very exciting result. We know that the lower bound for the *problem* of selection is $\Omega(n)$ — linear time. So we now have an algorithm that is asymptotically optimal *on average*. Yes, there may be some inputs where things go badly, but most of the time this will be the fastest possible (up to a constant factor at least). It looks like quickSelect is living up to its name!

4 Randomization

There are some pretty big disadvantages to doing average-case analysis. The main problem is that we had to make two pretty big assumptions:

- Every permutation of the input array is equally likely
- Every permutation of the sub-array for each recursive call is *also* equally likely.

This second assumption is true when the first one is, but we didn't prove it because it's a little too difficult.

But the first assumption is definitely *not* true in many (most?) applications of this problem. Typical inputs that show up in sorting and selection problems are *almost sorted*, meaning that there are only a few elements out of order. So not every permutation of the input is actually equally likely. Even worse, almost-sorted inputs will actually hit the worst-case cost of the quickSelect1 because the pivots chosen will usually be at or near the beginning of the sorted order!

So average-case analysis tells us something powerful (most instances are easy), but really we'd like to move away from these assumptions we had to make and have an algorithm that usually behaves well on *any* input.

4.1 Randomized Algorithms

A *randomized algorithm* is one that uses some source of random numbers in addition to the actual input in order to ge the answer. Let's see how random numbers could be used to speed up the quickSelect algorithm.

Remember that the problem with average-case analysis was the assumptions we had to make on the distribution (i.e., the relative likelihood) of inputs. The basic idea of the randomized approach is that we replace these assumptions on the input distribution — which we cannot control — to assumptions on the distribution of the random numbers. Since we pick the random numbers, we have much more confidence over their distribution than whatever input might come along. So we should end up with algorithms that once again perform well on average, but regardless of what the input distribution looks like.

(The previous paragraph is packed with wisdom. Re-read it.)

Now a first idea at randomizing the quickSelect algorithm is to first "shuffle" (i.e., randomly permute) the input array before sending it to the quickSelect1 algorithm. This way, we really know that each of the input orderings is equally likely.

And actually, this idea really works! It does selection in *expected* $\Theta(n)$ time. Notice that it's "expected" as opposed to "average". The distinction is a fine one but one worth understanding. Formally, the expected cost is defined similarly to the average cost: the sum of the probability of each possibility, times the cost of that possibility. It's just like the expected value of a random variable. (Remember those from Discrete?) Usually, we can figure this out by dividing the possibilities into some equally-likely cases (say m), summing the costs in each of those m cases, and dividing by m to get the expected cost. Don't worry, we'll see some examples.

4.2 "Stupid" Randomization

Remember that the main problem in the average-case analysis was our assumption that all permutations of the input order were equally likely. This is a bad assumption to make, especially if we want our algorithm or program to be useful in a variety of different applications.

But with randomization, we can shift the burden of this assumption from the input (which we can't control) to the random numbers we pick. First, we will have to assume there is a procedure random(n) which takes any integer n and returns any integer between 0 and $n-1$, with equal probability. For example, a call to random(5) would have a 20% chance of returning 3.

The following algorithm can be used to randomly re-order any array:

```
def shuffle(A):
    n = len(A)
    for i in range(0, n):
        swap(A, i, randrange(i, n))
```

Because the value of randrange(i,n) is just a random integer between i and $n-1$, this is repeatedly swapping each array element with some element that comes after it. At the end, each of the $n!$ permutations of the original input is equally likely.

Now this gives a really simple, 2-step randomized algorithm for the selection problem:

```
def randomSelect(A, k):
    shuffle(A)
    return quickSelect1(A, k)
```

This algorithm now has *expected worst-case running time* $\Theta(n)$, using exactly the same analysis as we did for the average-case of quickSelect1 before. The question you should be asking yourself is, why is this better? How could it possibly be of any benefit to randomly shuffle the array before running the same old selection algorithm?

The answer comes when we try to think of the worst case. We said that the worst-case input for quickSelect1 was when the input array A was already sorted. In this case, all the pivots will be really bad, the partitions will be totally lopsided, and the running time will be $\Theta(n^2)$.

But what's the worst-case input for RandomSelect? If you think about it, you should see that there really isn't any! The cost doesn't depend on the order of the input array A, just on the order that A gets "shuffled" into in the first step. In other words, *there are no unlucky inputs, only unlucky random numbers.* And since we know for sure that the shuffle algorithm randomly permutes the order of the elements in the array, the expected cost of the algorithm is the same no matter what the input is! This is why we can say the *worst-case* expected cost is $\Theta(n)$.

4.3 Better Randomized Selection

Actually there's a simpler way to randomize the quickSelect algorithm: randomly choose the pivot element instead of the first element in the array. Here's what it looks like:

```
def choosePivot2(A):
    # This returns a random number from 0 up to n-1
    return randrange(0, len(A))

def quickSelect2(A, k):
    swap(A, 0, choosePivot2(A))
    # Everything else is the same as quickSelect1!
    # ...
```

It should be obvious that the expected analysis will be the same as the average-case analysis we did of quickSelect1, with the very crucial difference that *no assumptions on the input are required.* Since the pivot is chosen randomly from all the array elements, the probability that we have a "good pivot" is at least $\frac{1}{2}$, and everything falls through like above to show that the worst-case expected cost is $\Theta(n)$.

Don't forget the big picture here. This is really exciting because we already figured out that $\Theta(n)$ is a lower bound on the cost of the selection problem, but the best algorithms we could come up with using heaps and sorting were still $O(n \log n)$ in the worst case (selecting the median). In fact, this randomized quickSelect algorithm (or something like it) is the fastest way to solve the selection algorithm in practice.

5 Median of Medians

We now have an excellent solution to the selection problem with the quickSelect2 algorithm. But there is still a nagging question: was randomization really necessary to get linear time? Is there an asymptotically optimal solution to this problem that doesn't require "changing the rules" and using average-case or randomized analysis?

To tackle this challenge, we are going to use the same basic "quickSelect" structure, but modify the pivot selection part once again. Remember that the very best pivot would be the median. But of course that's the actual problem we're trying to solve! So we will have to settle for an algorithm that chooses "good" pivots.

5.1 Pivot Selection

The basic idea is to choose the pivot using a divide-and-conquer algorithm. This is called the "median of medians" algorithm for reasons that will become clear really soon.

This algorithm depends crucially on a certain *parameter*, which we'll call q. This has to be an integer greater than 1, but we'll have to wait until the analysis to see exactly what it should be. Once we've chosen this constant q, the divide-and-conquer approach looks like this:

1. Split the input into size-q sub-arrays
2. Find the median of each sub-array
3. Make a sub-array containing the medians from step 2
4. Return the median out of the sub-array of medians

This is similar to other divide-and-conquer algorithms we've seen, such as MergeSort, but there at least one very important difference. There are no (directly) recursive calls! This is because the median-choosing required on steps 3 and 4 is accomplished by calls to the quickSelect algorithm, not recursive calls to the choosePivot algorithm.

But of course the quickSelect algorithm is going to call the choosePivot algorithm that we're trying to develop here! This crazy situation is known as *mutual recursion*, and it happens when two (or more) functions make nested calls to the other one. It means two things for us. First, we have to be careful that the size of the problem keeps getting smaller, so that we don't have an infinite recursive loop (that would be bad). Second, we will have to analyze both of the algorithms at the same time, since we won't be able to just analyze one in isolation and plug its cost into the other one.

Put all this together and this is what you get:

```
def choosePivot3(A, q=5):
    '''q is a parameter that affects the complexity; can be
       any value greater than or equal to 2.'''
    n = len(A)
    m = n // q
    if m <= 1:
        # base case
        return n // 2
    medians = []
    for i in range(0, m):
        # Find median of each group
        medians.append(quickSelect3(A[i*q : (i+1)*q], q//2))
    # Find the median of medians
    mom = quickSelect3(medians, m//2)
    for i in range(0, n):
        if A[i] == mom:
            return i

def quickSelect3(A, k):
    swap(A, 0, choosePivot3(A))
    # Everything else is the same as quickSelect1!
    # ...
```

5.2 An Example

Let's consider an example to see how this crazy algorithm actually works. Just for the purpose of this example, let the parameter $q = 3$, and let's see what pivot gets chosen for the following array:

$$A = [13, 25, 18, 76, 39, 51, 53, 41, 96, 5, 19, 72, 20, 63, 11]$$

You can see that A has size $n = 15$ and since q is 3 this means that the value computed on the first step is $m = 5$. The first step is to divide into m sub-arrays of size q, which means 5 sub-arrays of size 3:

$$[13, 25, 18], [76, 39, 51], [53, 41, 96], [5, 19, 72].[20, 63, 11]$$

The we find the median element in each sub-array: 18, 51, 53, 19, 20. Now the actual algorithm does this by making recursive calls on each sub-array and then moving each median to the front. After the first iteration of the **for** loop in choosePivot3 the array will look like

$$[18, 13, 25, 76, 39, 51, 53, 41, 96, 5, 19, 72, 20, 63, 11]$$

Where the first median 18 has been moved to the front of the array. Then after the second iteration it will be

$$[18, 51, 25, 39, 13, 76, 53, 41, 96, 5, 19, 72, 20, 63, 11]$$

Notice that now 18 and 51 have been moved to the front. After all 5 iterations of the **for** loop we get

$$[18, 51, 53, 19, 20, 76, 41, 25, 96, 5, 39, 72, 11, 13, 63]$$

The last call to quickSelect3 finds the median of the first 5 numbers (the medians), which is 20 for this example. So 20 is the pivot that will be returned to use in the partitioning for quickSelect.

5.3 Lopsidedness

That's a lot of work just to choose one pivot element! Remember that this will ultimately just be the first step in some call to quickSelect3, and in fact this whole process will get repeated many times through all the recursive calls.

The crucial question in figuring out how fast this complicated scheme will be is how "good" is the pivot that gets chosen? Look at the example above. The chosen pivot is 20, which will partition with 5 elements on the left (18, 19, 5, 11, and 13) and 9 on the right (51, 53, 76, 41, 25, 96, 39, 72, and 63). So it's not perfectly balanced. But how bad is this?

Let's think about how lopsided it could possibly be for a size-15 array with $q = 3$, like the example above. First, consider how small the left side of the partition could be, i.e., the number of elements less than the pivot. We know that the pivot is the median of the 5 medians, so it's definitely greater than 2 of the other medians (18 and 19 in this example). And each of those 2 and the pivot element itself are the medians of their individual sub-arrays, so they are each greater than 1 other element (13, 5, and 11 in this example). This means that *in the worst case*, no matter what, the pivot will be greater than at least 5 other elements in the array.

We can follow the same logic to say that, again for $n = 15$ and $q = 3$, the pivot will always be less than at least 5 other elements in the array. The example above is indeed a worst-case example: 5 elements on one side and 9 on the other is as bad as it can get. All that work really might pay off after all!

To generalize this, we know that the pivot will always be greater than or equal to exactly $\lceil m/2 \rceil$ of the medians, and each of those is greater than or equal to at least $\lceil q/2 \rceil$ elements in the array. Now since $m = \lfloor n/q \rfloor$, this works out to saying that the pivot will always be greater than or equal to, *and* less than or equal to this number of elements in the array:

$$\left\lceil \frac{n}{2q} \right\rceil \cdot \left\lceil \frac{q}{2} \right\rceil$$

Since we're interested in the worst-case analysis of the quickSelect algorithm that's ultimately going to come out of this, what we really care about is the largest possible size of the sub-array for a recursive call, which will be the larger of the two sides of the partition. This is a formula that I don't feel like writing over and over again, so I'll define it as $p_q(n)$:

$$p_q(n) = n - \left\lceil \frac{n}{2q} \right\rceil \cdot \left\lceil \frac{q}{2} \right\rceil$$

To confirm that this works with our example above, we can check that $p_3(15) = 15 - 3 * 2$, which is 9, the size of the larger side of the partition.

5.4 Analysis

What we are really interested in is the worst-case cost of the quickSelect3 algorithm. Writing a recurrence for this isn't too bad; it's just $T(n) = n + T(p_q(n)) + S(n)$. The n is from the cost of partitioning, the $T(p_q(n))$ is the worst-case cost of the recursive call to quickSelect3 (using the function we defined above), and finally $S(n)$ is the cost of the median-of-medians algorithm choosePivot3.

In order to analyze this, we will need to know the cost of the choosePivot3 algorithm. This is of course just another recurrence: $S(n) = n + mT(q) + T(m)$. The n is just the overhead from the loops and swapping etc., $mT(q)$ is the cost of finding each of the medians-of-q in the for loop, and $T(m)$ is the cost of finding the median-of-medians at the end.

Here's the trickiness: in order to find $T(n)$ we need to know $S(n)$, and in order to find $S(n)$ we need to know $T(n)$. This is the mutually-recursive structure of the algorithm wreaking havoc in our analysis. Fortunately, there's an easy fix in this case, and it comes from the fact that choosePivot3 does *not* make any recursive calls to itself. We just substitute the formula for $S(n)$ into the one for $T(n)$ and eliminate S from the picture entirely! Doing this, and using the fact that $m \in \Theta(n/q)$, we get:

$$T(n) = n + T(p_q(n)) + \frac{n}{q}T(q) + T(\frac{n}{q})$$

That's a nasty-looking recurrence for sure, but at least it just contains references to a single function $T(n)$. To actually analyze this function will require us to (finally) pick a value for q. But first we need a technical definition.

5.5 At Least Linear

The analysis of the quickSelect3 algorithm is ultimately going to have to combine those three references to $T(n)$ on the right-hand side of the recurrence. In order to do this, we will have to use the fact that we know the running time is more than linear. But it's actually a little bit more technical than that:

> **Definition**: At Least Linear
>
> A function $f(n)$ is *at least linear* if the related function $f(n)/n$ is non-decreasing (i.e. it gets larger or stays the same) as n increases.

For a very easy example, we know that $f(n) = n^3$ is "at least linear" because $f(n)/n = n^2$ is a non-decreasing function. In fact, any $f(n) = n^c(\lg n)^d$ with $c \geq 1$ and $d \geq 0$ is at least linear, as you might expect.

Here's why we care, and why this definition will allow us to do the analysis demanded by the complicated quickSelect3 function.

Lemma

If the function $T(n)$ is at least linear, then $T(m) + T(n) \leq T(n+m)$ for any positive-valued variables m and n.

Here's the proof: Re-write the left hand side as $m \cdot (T(m)/m) + n \cdot (T(n)/n)$. We know from the fact that T is at least linear that both $T(m)/m$ and $T(n)/n$ are less than or equal to $T(m+n)/(m+n)$. This means that:

$$T(m) + T(n) \leq m\left(\frac{T(m+n)}{m+n}\right) + n\left(\frac{T(m+n)}{m+n}\right) = (m+n)\left(\frac{T(m+n)}{m+n}\right) = T(m+n)$$

Which is what the lemma claims.

5.6 Choosing q

Now we have all the tools necessary to do the analysis. We just have to plug in some values for q to see what will work best.

Just to start thinking about this problem, consider what would happen if we chose $q = 1$. Then $m = n$, so the final call would just be selecting the median of the whole array — an infinite loop! Choosing $q = n$ would do the same thing; the first (and only) iteration through the **for** loop in the choosePivot3 algorithm would make a call to quickSelect3 on the entire array, and we have another infinite loop. So we definitely need to find a q that is somewhere between these extremes.

- First try: $q = n/3$

  This means splitting into 3 big subproblems, then choosing the median of those three medians. To do the analysis, we first have to figure out how big the larger side of the partition will be in the worst case:

  $$p_{n/3}(n) = n - \left\lceil \frac{n}{2n/3} \right\rceil \cdot \left\lceil \frac{n/3}{2} \right\rceil = n - 2\lceil n/6 \rceil \in \Theta(\frac{2n}{3})$$

  Now plug this into the recurrence for $T(n)$ and we get:

  $$T(n) = n + T(2n/3) + 3T(n/3) + T(3)$$

  We can drop the $T(3)$ term because that just means finding the median of three elements, which is constant-time. But what remains is still not good at all. What you should notice is that certainly $T(n) > n + 3T(n/3)$, but applying the Master Method to this gives $T(n) \in \Omega(n \log n)$, which is just a *lower bound* on the cost of this algorithm. So certainly if we choose a big value for q like $n/3$, we are not going to get a fast selection algorithm.

- Second try: $q = 3$

  This means splitting into a much of very small subproblems (of size 3), and then just one large subproblem (of size $n/3$). This corresponds to the example above. Again, we start by figuring out:

  $$p_3(n) = n - \left\lceil \frac{n}{6} \right\rceil \cdot \left\lceil \frac{3}{2} \right\rceil = n - 2\lceil n/6 \rceil \in \Theta(2n/3)$$

  Same as before! But the overall analysis is certainly not the same:

  $$T(n) = n + T(2n/3) + (n/3)T(3) + T(n/3)$$

  Now something wonderful happens: the middle term goes away! This is because $T(3)$ is the cost of finding the median of 3 elements, which is just a constant, and therefore $(n/3)T(3) \in \Theta(n)$, and it gets taken care of by the n that's already in the formula.

  So we have $T(n) = n + T(2n/3) + T(n/3)$. This seems like it should be better than before, but unfortunately it's still not a fast selection algorithm. Here's proof: Use the fact that $T(n)$ is "at least linear" to conclude

that $T(2n/3) \geq T(n/3) + T(n/3)$. Plugging this into the recurrence gives $T(n) \geq n + 3T(n/3)$. Now just like before (oh no!), this is $\Omega(n \log n)$ — not a fast algorithm for the selection problem.

So $q = 3$ is no good either.

- Third try: $q = 4$

  Third time's a charm right? Following the same process as for $q = 3$, we get $p_4(n) \in \Theta(3n/4)$, and therefore

  $$T(n) = n + T(3n/4) + (n/4)T(4) + T(n/4)$$

  Once again the middle term drops out ($T(4)$ is a constant too) and we are left with $T(n) = n+T(3n/4)+T(n/4)$. But using the same reasoning as before we get $T(n) \geq n + 4T(n/4)$, which once again is $\Omega(n \log n)$.

  Are you ready to give up yet?

- Fourth try: $q = 5$

  Let's go through the same motions like with 3 and 4. First we determine that $p_5(n) \in \Theta(7n/10)$. Were you expecting $4n/5$? Well it's not! Check the formula if you don't believe me!

  Plugging this into the recurrence gives us

  $$T(n) = n + T(7n/10) + (n/5)T(5) + T(n/5)$$

  The middle term goes away just like before, but what we are left with is not so simple: $T(n) = n + T(7n/10) + T(n/5)$

  Here we can finally use the "at least linear" condition to *help* the analysis: Because the function T must be at least linear, we know that $T(7n/10) + T(n/5) \leq T(7n/10 + n/5)$. This simplifies to just $T(9n/10)$. So we have $T(n) \leq n + T(9n/10)$.

  Finally something that we can really get a big-O bound on! Using the Master Method (with $a = 1, b = 10/9, c = 1, d = 0$), we see that $e = \log_{10/9} 1 = 0$, which is less than c, so the total cost is $T(n) \in O(n)$.

  I will pause while you reflect on this momentous achievement.

5.7 Conclusions

Giving where credit where it is certainly due, this *worst-case linear-time selection algorithm* was invented by (appropriately) 5 computer scientists, Blum, Floyd, Pratt, Rivest, and Tarjan, in 1973.

And where is this brilliant algorithm used in practice? Well, not really anywhere. The algorithm is indeed $O(n)$ in the worst case, but it is a very complicated algorithm (much more so than the other quickSelects), and the "hidden constant" in front of n in the true cost is quite large. For all practical purposes, you should probably use the quickSelect2 algorithm.

What has been accomplished is that we've learned something about computers and about the world. In particular, we've learned that randomization is *not* required in order to solve the selection problem in linear time. And hopefully we've also picked up some tricks and techniques for algorithm design and analysis that will be useful elsewhere.

In summary: yes, this is a "theoretical" result. There is a strong theory component to algorithms, and we will see more of it as we go along. Not everything in algorithms translates to the fastest possible computer program, and *that's okay.*

6 QuickSort

The same idea that we used for all the variants of the quickSelect algorithm can also be applied to the sorting problem. Just like before, we start by choosing a pivot element and then calling partition to sort based on that single element. But unlike with quickSelect, for sorting we will always need two recursive calls, for each side of the partition. Here's what the algorithm looks like:

```
def quickSort1(A):
    n = len(A)
    if n > 1:
        swap(A, 0, choosePivot1(A))
        p = partition(A)
        A[0 : p] = quickSort1(A[0 : p])
        A[p+1 : n] = quickSort1(A[p+1 : n])
    return A
```

As you might have guessed based on its relative simplicity, the quickSort algorithm was actually invented before quickSelect, by C.A.R. Hoare all the way back in 1960. And it is an eminently practical algorithm, one of the most commonly-used general-purpose sorting algorithms in practice.

Of course, we know it can't be any faster *asymptotically* than MergeSort and HeapSort, since they are both asymptotically optimal in the comparison model. The advantage of quickSort is that it generally requires fewer swaps and comparisons than HeapSort, but always requires much less memory than MergeSort. But of course it's not enough just to say this. We need to do some analysis!

6.1 Best-case, Worst-case

The worst case of quickSort is very much like that for quickSelect. If the chosen pivot is the smallest or largest element in the array, then the partition will be very lopsided, and one of the recursive calls will be on a sub-array of size $n-1$. The worst-case cost, as with quickSelect1, is $\Theta(n^2)$.

The best case, however, is different. Since the algorithm always performs two recursive calls whenever $n > 1$, the best case is that the chosen pivot is always a median of the array. Then the running time is given by the recurrence $T(n) = n + 2T(n/2)$, which is the same as MergeSort, $\Theta(n \log n)$.

So once again we have an algorithm whose best-case time is pretty good, but worst-case is not so good, and we want to argue that the algorithm is still very effective "most of the time". What do we need?

6.2 Average-case analysis

The average-case analysis of quickSort1 will use many of the same ideas from that of quickSelect1, with some significant differences of course. Again we start by assuming (a BIG assumption!) that every ordering of the input is equally likely.

And once again the position of the pivot element is pretty important. If the chosen pivot lands at position p in the sorted order, then the two recursive calls are of size p and $n-p-1$. Since each permutation is equally likely, and choosePivot1 just picks the first element as the pivot, each of the n positions for that pivot are also equally likely. Therefore we get the following recurrence for the average cost of quickSort1:

$$T(n) = n + \frac{1}{n}\sum_{p=0}^{n-1}\left(T\left(p\right) + T\left(n-p-1\right)\right)$$

You can look in the textbook to see a detailed analysis of this complicated recurrence. But there is once again a simpler way to get the same (asymptotic) result.

Remember the good pivots and bad pivots? Let's use that same idea again. Recall that a "good pivot" is one whose position is in the middle half of the sorted order, so $n/4 \le p < 3n/4$. So the most unbalanced "good pivot" will result in one recursive call of size $3n/4$ and one of size $n/4$. The most unbalanced "bad pivot" will result in one recursive call of size 0 and one of size $n-1$. This is summarized as follows:

$$T(n) \le \begin{cases} n + T(\frac{3n}{4}) + T(\frac{n}{4}), & \text{good pivot} \\ n + T(n-1) + T(0), & \text{bad pivot} \end{cases}$$

Because $T(0)$ is just a constant, we can simplify the "bad pivot" case to just $T(n) \le n + T(n)$.

Now since each of these possibilities has probability $\frac{1}{2}$, the average cost is

$$T(n) \leq \frac{1}{2}\left(n + T(3n/4) + T(n/4)\right) + \frac{1}{2}\left(n + T(n)\right)$$

You should recognize how to simplify this: multiply both sides by 2, subtract $T(n)$ from both sides, and combine like terms to get $T(n) \leq 2n + T(3n/4) + T(n/4)$.

Unfortunately, this isn't quite simple enough to apply the Master Method just yet. To do it, we'll use our old friend the "at least linear" lemma from before. Because the function T is at least linear, we know that $3T(n/4) \leq T(3n/4)$. Dividing both sides by 3, $T(n/4) \leq \frac{1}{3}T(3n/4)$. Now substitute into the recurrence above and we get

$$T(n) \leq 2n + \frac{4}{3}T(\frac{3n}{4})$$

Now finally we can apply master Method A, with $a = 4/3, b = 4/3, c = 1, d = 0$. This gives $e = \log_b a = 1$, which equals c, so the formula is asymptotically $O(n \log n)$. Since the average-case can't possibly be better than the best-case, we conclude that the average cost of quickSort1 is $\Theta(n \log n)$.

6.3 QuickSort variants

QuickSort can be improved from this "version 1" just like QuickSelect was. Using the randomized choosePivot2 subroutine, the resulting randomized quickSort2 algorithm has worst-case expected running time $\Theta(n \log n)$, using the same analysis as above. This is the best way to do QuickSort for practical purposes, and it eliminates the potential for "unlucky input" like before.

And we can also use the median-of-medians algorithm to get a non-randomized, asymptotically optimal version. Using the choosePivot3 method (mutually recursively with quickSelect3) to choose pivots, the resulting quickSort3 algorithm has worst-case running time $\Theta(n \log n)$, which of course is the best possible. And just like before, the overhead involved in this approach prevents it from being useful in practice.

7 Sorting without Comparisons

So far we have seen a number of sorting algorithms (MergeSort, HeapSort, QuickSort) that are *asymptotically optimal*, meaning their worst-case running time is the same as the lower bound for the problem of sorting we proved earlier, $\Theta(n \log n)$.

But remember that this lower bound was in the *comparison model* only; it required the assumption that an algorithm's only information about the relative order of the input elements comes from comparing pairs of them. It's most common to view this as an impossibility: we *can't* sort faster than $n \log n$ in the comparison model. But we can also view the lower bound as a guide towards improvement: by doing something other than pairwise element comparisons, we might be able to sort faster than $n \log n$.

We'll see a few approaches that actually achieve this. The key to success for each of them is some assumption on the nature of the input elements that isn't true in the most general case. So the best we'll be able to say is that in some special circumstances, we can sort faster than $\Theta(n \log n)$ time.

7.1 Bucket Sort

When I'm sorting a pile of assignments to hand back, I usually start by putting them into four or five piles according to the first letter of the student's last name. The piles might look like "A-D", "E-K", "L-P", "Q-Z". It's okay if all the piles aren't exactly the same size. Then afterwards, I sort each pile with something like a selection sort, a much easier task than sorting the all of the assignments at once. Finally, the sorted piles are stacked in order.

This is an example of a general approach called "bucket sort" or "bin sort", and by now you should recognize the general design paradigm here: divide and conquer. Here's a more formal outline of this approach:

1. Split the range of elements into k subranges or "buckets"
2. Iterate through the array and insert each element into the appropriate bucket
3. Sort each bucket (somehow — perhaps recursively)
4. Concatenate the sorted buckets, in order, to the output

Have we seen an algorithm like this before? Yes - QuickSort! The partition-then-recurse outline of QuickSort is like doing a BucketSort with 2 buckets. This demonstrates how Bucket Sort is really a general way of approaching the sorting problem, and isn't actually a specific algorithm. (We can't analyze it, for instance.)

Furthermore, even though QuickSort is a type of Bucket Sort, the real benefit to bucket sorting comes from having multiple buckets, and assuming that *we can tell instantly (in constant time) which bucket each element goes in.* So when I'm sorting your assignments, I don't have to check each one against each bucket; I just look at the letter and put it into the corresponding pile. This is something that we can't do in the comparison model, and it's why bucket sort is the starting point in our quest to sort faster than $\Theta(n \log n)$ time.

7.2 Counting Sort

For some sorting problems, the number of different keys in the array might be even smaller than the size of the array itself. This restricted setting is the basis for counting sort.

Consider again my terribly-difficult problem of sorting assignments before handing them back. Maybe instead of sorting by names, I just want to sort by the letter grade received (A, B, C, D, or F). So I should do a bin sort with 5 bins, one for each possible grade. But notice that, once I know which bin a paper goes in, I'm done! There is no need to sort the bins themselves, since every paper in that bin has the same grade.

Counting Sort works similarly, for arrays in which the keys are limited to a certain small range of values. For simplicity, we'll say each key in the array is an integer between 0 and $k - 1$, for some integer k. Then the sort works in two stages:

1. **Count** how many keys of each value are present. Store these counts in an array C (the "counting array").
2. Make one more pass through the array, and **place** each element into its proper position in the sorted output array, according to the counts.

In between these two stages, the positions of the elements with each key are determined by computing partial sums of the counts. Let's examine a concrete example to see how this works.

Back to my problem of sorting papers by their grades, let's say I have the following grades and the names that go along with them:

Name	Grade
Bill	A
Wendy	F
Bob	C
Vicki	C
Brian	B
Luke	C
Susan	A
Mary	B
Steve	A

The first thing to do is go through the array and count how many of each grade there are:

Grade	Count
A	3
B	2

Grade	Count
C	3
D	0
F	1

Now I want to compute the position where each element will go. So for example the A's will be in positions 0,1,2, the B's in positions 3,4, the C's in positions 5,6,7 the D's in no positions (because there aren't any), and the F in position 8. In general, *the starting position of each key is the sum of the counts of the previous keys.* So for example the C's start at position 5 because the count of A's plus the count of B's is $3 + 2 = 5$.

Now once, I know these positions, taking one more pass through the original array and putting each element into its proper position in the output is a simple matter.

Here is the formal description of this algorithm:

```
def countingSort(A, k):
    C = [0] * k # size-k array filled with 0's
    for x in A:
        C[x] = C[x] + 1
    # Now C has the counts.
    # P will hold the positions.
    P = [0]
    for i in range(1, k):
        P.append(P[i-1] + C[i-1])
    # Now copy everything into its proper position.
    for x in copy(A):
        A[P[x]] = x
        P[x] = P[x] + 1
    return A
```

The analysis is pretty straightforward. There are three loops: to caluclate the counts, to calculate the initial positions, and to do the final placement. The first and last loops are both $\Theta(n)$ time and the second loop is $\Theta(k)$, for a total cost of $\Theta(n+k)$ in every case.

Notice that k plays an important role in the cost! In particular, if $k \in O(n)$, then the total cost of the algorithm is $O(n)$ — linear-time sorting!

We should also observe the cost in terms of space. For this function, there are two new arrays allocated, C and B. Their total size is $k+n$, so the space cost of the algorithm is also $\Theta(n+k)$.

7.3 Stable Sorting

One very important property of CountingSort is that it is a *stable* sort, which means that the relative ordering of elements with the same keys is preserved from the original array.

In the example above, Bill, Susan, and Steve all have A's, and they show up in that order in the original array. So they will be the first three names in the sorted array. But if we did not have a stable sort, then they might show up like "Susan, Bill, Steve" or "Steve, Bill, Susan" — the relative order between them doesn't matter since they all have the same sorting key of "A". But since CountingSort is *stable*, we know that they will end up in the order Bill, Susan, Steve in the final sorted array, since that is the order they appear in in the original array.

Now we can go back and examine the other sorting algorithms to see whether they are stable. MergeSort can easily be made stable if ties in the "merge" step always go to the element in the first array. SelectionSort and InsertionSort are also pretty easy to make stable. But HeapSort and QuickSort are not; these are inherently *unstable* sorts. The problem is the bubbleDown and partition subroutines in these algorithms, that will move elements in the array around and totally destroy the original relative order.

In fact there is a general trend: *stable sorts use more space, or more time.* Remember that a big benefit of both QuickSort and HeapSort is that they work "in-place", without having to allocate extra arrays. The price to pay for

that space savings, apparently, is the loss of stability. Incidentally, QuickSort can be made stable if the partition algorithm is changed to work on a copy of the array rather than swapping elements in the original array. Can you think of how to do this?

7.4 Radix Sort

Okay, let's go back to the stack of graded papers example one more time. Maybe we want to sort them by grade, but for students that have the same grade, we want to sort them by their names. So we want to sort primarily by the grade, and secondarily by the name. (And maybe the name is primarily by the last name, secondarily by the first name, and so on.) How can we do this?

One idea (using the divide-and-conquer mentality) is to first sort into groups based on the primary key (grade), and then sort each group separately (by name). This works fine in many cases.

But a simpler *program* will result if we go in the other direction and use stable sorts. So we first sort the whole list by the least-significant key (maybe first name), using a stable sort, then sort by last name using another stable sort, and finally by grade, again using a stable sort. Same result as before! In fact, this is how sorting something like a spreadsheet actually works.

There is a general algorithm based on this principle, called *radix sort*. For simplicity, I'm presenting it as sorting a list of n numbers, each of which has d digits. So we sort digit-by-digit, starting with the least significant digit. And what should we use to sort by each digit? CountinSort is perfect for the task, because it is a stable sort and it relies on the numbers (digits now) being restricted to a certain range. Here it is:

```
def radixSort(A, d, B):
    for i in range(0, d):
        countingSort(A, B) # based on the i'th digits
    return A
```

Each CountingSort costs $\Theta(n+B)$, so the total running time of RadixSort is $\Theta(d(n+B))$. In many applications the base B is a constant (like 10), so this is actually just $\Theta(dn)$. Since it doesn't follow the comparison model, it once again has the potential to be faster than $\Theta(n \log n)$. Exciting!

8 Summary

Here are the sorting algorithms we have seen, with some informative comparisons. Note that *all* of these algorithms are useful for some purpose or another!

Algorithm	Analysis
SelectionSort	$\Theta(n^2)$
InsertionSort	$\Theta(n)$ best, $\Theta(n^2)$ worst
HeapSort	$\Theta(n \log n)$
MergeSort	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$, depending on pivoting
CountingSort	$\Theta(n+k)$
RadixSort	$\Theta(d(n+B))$

We also learned about the selection problem and its variants (QuickSelect 1, 2, and 3, including the Median of Medians algorithm). Along with this came some new and exciting analysis techniques as well: average-case and randomized analysis, and the "at least linear" definition and lemma for simplifying sums of recursive terms.

9 Back to Kruskal

A few units ago, we learned Kruskal's algorithm for finding the minimum spanning tree in a graph. The gist of the algorithm is as follows:

1. Sort the edges by increasing order of edge weight 2 Initialize a disjoint-set data structure with every node in a separate set
2. Pick out the next-smallest edge in the list.
3. If that edge connects two vertices that aren't already connected, then add the edge to the MST and perform a union operation in the data structure to join the vertices connected on either end.
4. Repeat steps 3 and 4 until you run out of edges (or until everything is connected).

For a graph with n vertices and m edges, running this algorithm costs $O(n + m \log m)$ time. The expensive, dominating step is (1), sorting the edges by weight.

Well we've just seen some faster ways to sort. That's exciting, because it gives hope that MST can be computed faster also. For example, in a graph where the largest edge weight is $O(m)$, we could use a simple counting sort to sort those edges in $O(m)$ time. That would reduce the total cost of Kruskal's algorithm to just $O(m + n \log n)$.

And now we can think about improving the data structure, because the $O(n \log n)$ cost of doing n union operations on the disjoint-set data structure is dominating the whole complexity of the algorithm.

The trick here is to use a different storage for the disjoint-set data structure. Instead of storing each set as a linked list, with the head of the linked list counting as the "name" of that set, you store each set as a *tree*. Each item in the set points back towards the root of the tree, and the name of the root counts as the name of the entire set. So if you want to do a find operation to determine which set an element is in, you just follow the pointers up to the root node and return the root node's label.

The big improvement comes from doing what's called "path compression" every time find gets called. As you're traversing up the tree to the root node, you "compress" the path by making every node in the path point directly to the root of the tree. That ensures that all the trees stay really *really* flat, almost constant height in fact.

Well actually the height is determined by what's called the Inverse Ackermann function, which is one of the slowest-growing functions we know about, much slower than $O(\log n)$ or even $O(\log \log \log \log n)$. We write this function as $\alpha(n)$, and to give yuou an idea of just how slow-growing we mean, $\alpha(2^{2^{2^{64436}}}) \approx 4$. So for all practical purposes, this is a constant.

So take my word for it (or read your textbook!) that if we apply path compression to the fast union-find data structure we get $O(n\alpha(n))$ amortized cost to do all the union and find operations. So the total cost of Kruskal's can finally be improved to the measly $O(m + n\alpha(n))$, which is *very* close to, but not quite achieving, linear time.

And now I must tease you by telling you that there is in fact a linear-time algorithm for MST that actually has $O(n + m)$ running time, but you need randomization to get it! So it will have to wait for another class...